

Literatur und Software

- [1] Laurent Bourcellier, Jonathan Perez: Luciole, 2019, <https://www.luciole-vision.com/luciole-en.html> (besucht am 3. 4. 2020).
- [2] Britta Büchner u. a.: Die medizinische Diagnose »Legasthenie« ist irreführend und schadet den Interessen der Kinder, Stellungnahme zur Leitlinie: »Diagnostik und Behandlung von Kindern und Jugendlichen mit Lese- und/oder Rechtschreibstörung«, http://www.legakids.net/fileadmin/user_upload/Downloads/Info/Wissenschaft/LegaKids_Stellungnahme_Leitlinien_Mai_2015_01.pdf (besucht am 3. 4. 2020).
- [3] Herbert Voß: »Schriften für Menschen mit einer Leseschwäche«, *Die TeXnische Komödie*, 30.1 (2018), 54–56.

Verschiedene Möglichkeiten zum Kommentieren und Dokumentieren von Code

Lukas C. Bossert

Dieser Artikel ist Herbert Voß anlässlich seines Abschiedes aus der Redaktion der TeXnischen Komödie gewidmet. In der Redaktion sowie bei Seminaren oder auf TeX.SE hat er sicherlich nicht nur mir, sondern vielen anderen mit hilfreichen Kommentaren zum Code geholfen.

Selbst geschriebener Code ist Forschungsleistung: Eigene Gedanken und Ideen fließen in die Software/das Skript ein und sind daher mitunter erklärungsbedürftig. Diese Erklärungen zu liefern ist nicht immer eine leichte Aufgabe.¹ Dabei spielt es an sich keine Rolle, in welcher Programmiersprache der Code verfasst wird. Die Dokumentation ist hinsichtlich eines guten Forschungsdatenmanagements unerlässlich, wobei die Kommentare die Metadaten sind, die das notwendige Verständnis für die Forschungsdaten (Code) liefern.

Es gibt verschiedene Wege, diese Metadaten bereitzustellen. Dieser Beitrag zeigt drei Möglichkeiten dazu auf. Ausgehend von der einfachen Art anhand des Kommentarzeichens, dann mittels spezieller Pakete bis hin zu *literate programming*, als eine formvollendete Art Code zu beschreiben. Die Prinzipien des Kommentierens

¹ Viele hilfreiche Tipps und Anmerkungen zum Kommentieren von Code finden sich im äußerst lesenswerten Buch von *Weniger schlecht programmieren* von Passig und Jander, besonders die Seiten 61–76. Einige dieser Tipps sind in den Artikel eingeflossen. Andere sagen, es sollte nicht dokumentiert werden, was der Code tut, sondern vielmehr warum; siehe dazu [11, S. 243–248]. Manchmal ist das »Was« (speziell für Anfänger) ebenso wichtig.

und Dokumentierens können auf andere Sprachen übertragen werden, nachfolgend werden sie für L^AT_EX erläutert.

Für Anfänger: Dokumentieren in Kommentaren

Die schnellste und unkomplizierteste Form der Kommentierung von Code ist die mittels `%`. Dabei schreibt man nach dem Prozentzeichen seinen Kommentar, welcher dann vom Compiler ignoriert wird. Als T_EX-Anfänger habe ich diese Form des Kommentierens gern genutzt, um nach dem Aufruf des Pakets dessen Funktion zu notieren, oder die der einzelnen Optionen.

```

1 \usepackage[
2   left = 2cm, % Abstand links/innen
3   right = 4cm, % Abstand rechts/außen
4   top = 3cm, % Abstand oben
5 ]{geometry} % Paket kümmert sich um Seitenmaße etc.
```

Sobald man anfängt, größere Code-Blöcke zu tippen, die man beispielsweise aus Internetforen hat, ist es äußerst hilfreich, sich die Quelle zu notieren, damit man gegebenenfalls dort nochmals nachschauen kann.

Listing 1: Beispiel entnommen von <https://tex.stackexchange.com/a/524900/98739>.

```

1 %COMMAND FOUND ON INTERNET AND WORKS
2 % \printlist[<sep>]{<list macro>}
3 \newcommand{\printlist}[2][,]{% Print list
4   % http://tex.stackexchange.com/a/89187/5764
5   \def\listsep{\def\listsep{#1}}% Delayed execution of list separator
6   \renewcommand{\do}[1]{\listsep`##1'}%
7   [\dolistloop\languagelist]
8   }}
```

Ebenso ist es hilfreich, wenn man einen neuen Befehl mit Argumenten einführt, die Belegung der Argumente in einem Kommentar festzuhalten.

```

1 %% Argumentbelegung:
2 %% 1: [optional] Farbe
3 %% 2: {pflicht} Schlagwort
4 %% 3: {pflicht} Erklärung
5 %% 4: {pflicht} Schlusszeichen
6 \newcommand\MeinNeuerBefehl[3][black]{%
7   \bgroup
8   \color{#1}
9   \textbf{#2}: #3#4
10  \egroup
```

```
11 | }
```

Es gibt bei dieser Kommentiermöglichkeit ein kleines Caveat: Man sollte darauf achten, dass das Kommentarzeichen % nicht direkt ans Wortende gesetzt wird, da ansonsten ein notwendiges Leerzeichen »verschluckt« wird. Also lieber % erst nach einem vorangegangenen Leerzeichen setzen.

Es zeigt sich, dass der Gebrauch von %, speziell für T_EX-Neulinge sehr zu empfehlen ist: Man kann die wichtigen Informationen direkt beim auszuführenden Code unterbringen, um dort gegebenenfalls noch einmal nachzuschauen wie beispielsweise ein Befehl funktioniert. Inline- und mehrzeilige Kommentare haben unbestritten Vorteile und man sollte darauf nicht verzichten. [vgl. 9, S. 61–76]

Für Fortgeschrittene: Dokumentieren anhand von Paketen

Während die Nutzung von % mit anschließendem Kommentar besonders für das eigene Verständnis von Code hilfreich ist, braucht es eine andere Art der Dokumentation, wenn man einer Leserschaft Code präsentiert: Möchte man ausführbaren Code sichtbar in ein PDF setzen, um dessen Funktionsweise zu beschreiben, bedarf es einer anderen Herangehensweise.

Kurze Wörter oder Befehle kann man in einem Fließtext kommentieren und den Code mittels `\verb|befehl|` ausgeben: Alles, was zwischen den beiden gleichen Zeichen nach `\verb` steht, wird buchstabengetreu (verbatim) abgedruckt. Es findet in dieser Umgebung also keine Verarbeitung von Befehlen oder sonstigem Code statt. Für große Code-Blöcke ist diese Form der Kommentierung aber nicht geeignet. Dafür greift man besser zu Paketen, die darauf spezialisiert sind. Für Code-Darstellung sind besonders die Pakete `listings` [7], `minted` [10] oder `tcolorbox`² [13] verbreitet. Jedes Paket hat seine eigenen Vor- und Nachteile und es lohnt sich, in deren Dokumentationen einen Blick zu werfen.

Ich greife `listings` für ein Beispiel heraus. Der Code wird in eine `lstlisting`-Umgebung gesetzt. Davor oder danach kann man den Code dokumentieren:

² `tcolorbox` lädt selbst das Paket `listings`. Die Darstellungsweise von Code und der »Ausgabe« ist mit `tcolorbox` jedoch vielfältig modifizierbar, sodass es hier zusätzlich genannt wird.

Um eine Liste mit Aufzählungen zu erstellen, eignet sich die Umgebung `itemize`. Für jedes Aufzählungszeichen verwendet man `\item`. `\item` hat nur gegebenenfalls ein optionales Argument, dessen Inhalt als lokales Aufzählungszeichen übernommen wird. Das nachfolgende Beispiel erstellt eine Liste mit drei Punkten, wobei das letzte Aufzählungszeichen ein `!` ist.

```
\begin{itemize}
  \item \LaTeX
  \item \LuaTeX
  \item[!] \TeX
\end{itemize}
```

Um eine Liste mit Aufzählungen zu erstellen, eignet sich die Umgebung `\verb|itemize|`. Für jedes Aufzählungszeichen verwendet man `\verb|\item|`. `\verb|\item|` hat nur gegebenenfalls ein optionales Argument, dessen Inhalt als lokales Aufzählungszeichen übernommen wird. Das nachfolgende Beispiel erstellt eine Liste mit drei Punkten, wobei das letzte Aufzählungszeichen ein `\verb|!` ist.

```
\begin{lstlisting}
\begin{itemize}
  \item \LaTeX
  \item \LuaTeX
  \item[!] \TeX
\end{itemize}
\end{lstlisting}
```

Die genannten Pakete haben zusätzlich die Möglichkeit, Wörter des in der Umgebung stehenden Codes hervorzuheben, die bspw. typisch für die Programmiersprache sind. Dies lässt sich auf globaler oder lokaler Ebene, das heißt auch für jede `lstlisting`-Umgebung individuell, einstellen. Dies wird im optionalen Argument definiert.

```
\newcommand\myVerbatim[1]{%
  \texttt{#1}}
```

```
\begin{lstlisting}[basicstyle=\ttfamily,
  keywordstyle=\bfseries,
  language={\AllLaTeX}{TeX}]
\newcommand\myVerbatim[1]{\texttt{#1}}
\end{lstlisting}
```

Will man die Hervorhebung für \LaTeX jedoch innerhalb des gesamten Dokuments bestimmen, dann übergibt man in der Präambel die Sprache dem Befehl `\lstset`: `\lstset{language={\AllLaTeX}{TeX}}`

Der Befehl `\lstlistinginput{<Pfad/Datei.Endung>}` ist für die Code-Dokumentation von ganzen Dateien geeignet, anhand dessen der komplette Inhalt einer externen Datei buchstabengetreu in die PDF eingefügt wird. Diese Funktion ist hilfreich, wenn man einzelne Code-Blöcke der externen Datei kommentiert, die man passagenweise (`linerange=<Start>-<Ende>`) einfügt.

Im nachfolgenden Beispiel wird der Code einer fiktiven Dokumentenklasse beschrieben.

```

1 Zunächst werden alle neuen Dokumentenbefehle eingeführt.
2 \lstinputlisting[linerange=4-10]{my-class.cls}
3 und anschließend findet die Definition der Farbwelt
4 entsprechend dem Corporate Design statt:
5 \lstinputlisting[linerange=11-16]{my-class.cls}

```

Diese Vorgehensweise ist jedoch aufwendig, da sich der Code in der externen Datei ändern kann und man dann die Zeilenauswahl händisch nachjustieren muss.

So praktisch die `lstlisting`-Umgebung ist, eignet sie sich vor allem für kurze Code-Blöcke. Für die Dokumentation von Code, wie beispielsweise in Dokumentenklassen oder Bibliografie-Stilen, bedienen sich versierte Nutzer eines Vorgehens, das man in etwa mit *literarischem Programmieren* übersetzen kann. Davon handelt der nächste Abschnitt.

Für Experten: *Literate Programming*

»Documentation must be regarded as an integral part of the process of design and coding. A good programming language will encourage and assist the programmer to write clear, self-documenting code, and even perhaps to develop and display a pleasant style of writing.« [3, S. 195]

Diesem Ansatz hatte sich auch Knuth verschrieben, als er *literate programming* [4] für die Dokumentation von *The T_EXbook* [5] entwickelte.³ Ein Text, der auf angenehme Weise Code beschreibt.

Das Besondere von *literate programming* ist, dass innerhalb einer Datei den Programm-Code und gleichzeitig auch die Dokumentation enthalten ist. In zwei Schritten wird zuerst der Programm-Code extrahiert und in die entsprechende Datei geschrieben. Anschließend wird die Dokumentation erstellt, in der die einzelnen Programm-Code-Blöcke beschrieben sind.

Zwei Dateien werden in der Regel benötigt, um Programm-Code und Dokumentation zu erstellen: Die Datei mit dem dokumentierten Programm-Code (`.dtx`-Datei) und eine Datei, in der unter anderem Anweisungen enthalten sind, welche Code-Blöcke in welche Datei geschrieben werden sollen (`.ins`-Datei).

Der Clou bei diesem Verfahren ist, dass man die Zeilen des Programm-Codes direkt kommentieren kann, ohne dass man das Dokument wechseln oder den Programm-Code anschließend händisch in die entsprechende Code-Datei kopieren müsste.

³ Siehe die Sammlung an Texten zu *literate programming* in [6].

Dieses Vorgehen ist der de-facto-Standard für die Dokumentation von L^AT_EX-Paketen.⁴ Mittlerweile ist diese Form der Code-Dokumentation weit verbreitet und wird auch für andere Programmiersprachen verwendet.⁵

Die Dokumentation eines Beispiel-Pakets

Im folgenden Abschnitt wird anhand eines fiktiven (aber funktionierenden) Pakets namens *MeinPaket* die Funktionsweise des *literate programming* gezeigt.

Die Datei zum Paket *MeinPaket* heißt *MeinPaket.dtx*.⁶ In der Datei sind der eigentliche Code sowie die Dokumentation hinterlegt. Hier folgt zunächst einmal der Code, die Detailbeschreibung wird danach angegangen.

Listing 2: Dokumentation und Programm-Code des fiktiven Pakets *MeinPaket*.

```

1 % \iffalse
2 %<*batchfile>
3 \begingroup % startet eine Gruppe
4 \input docstrip.tex
5 \keepsilent % kein detailliertes Feedback
6 \askforoverwritefalse % überschreibe alte Version
7 \generate{\file{MeinPaket.sty}{\from{\jobname.dtx}{sty}}}
8 \endgroup % beendet die Gruppe
9 %</batchfile>
10 %<*driver>
11 \documentclass{ltxdoc}
12 \author{Lukas C. Bossert}
13 \title{Mein erstes \LaTeX-Paket}
14 \usepackage{MeinPaket} % lädt das neue Paket
15 \begin{document}
16 \maketitle % erstellt eine Titelseite
17 \DocInput{\jobname.dtx} % fügt sich selbst ein
18 \end{document}
19 %</driver>
20 %<*sty>
21 % \fi
22 %\section{Überblick}
23 % Mit dem Paket lässt sich die Schriftfarbe
24 % und Schriftgröße innerhalb einer Umgebung ändern.
```

⁴ Siehe etwa <https://www.ctan.org/help/upload-pkg>. Zum Thema Paketdokumentation (und zur Frage, ob diese auf deutsch oder auf englisch verfasst werden kann/soll), siehe [12, S. 32 f.].

⁵ <http://www.literateprogramming.com/index.html>

⁶ Der vollständige Code ist online hinterlegt: <https://gist.github.com/LukasCBossert>, dann unter *MeinPaket.dtx*

```

25 %\section{Code-Dokumentation}
26 % Als erstes wird die Grundvoraussetzung für das
27 % Paket spezifiziert,
28 % anschließend erfolgt die Namensnennung.
29 % \begin{macrocode}
30 \NeedsTeXFormat{LaTeX2e}[2005/12/01]
31 \ProvidesPackage{MeinPaket.sty}
32 % \end{macrocode}
33 % Es folgt der Code des Pakets. \par
34 % Das Paket "MeinPaket" lädt ein anderes Paket.
35 % \begin{macrocode}
36 \RequirePackage{xcolor}
37 % \end{macrocode}
38 % Es stellt auch eine eigene Umgebung zur Verfügung.
39 % \begin{environment}{MeinBeispiel}
40 % Innerhalb der Umgebung \meta{MeinBeispiel}
41 % wird die Schriftfarbe und -größe geändert.
42 % \begin{macrocode}
43 \newenvironment{MeinBeispiel}
44 % \end{macrocode}
45 % Zu Beginn wird die Schrift rot eingefärbt und
46 % vergrößert.
47 % \begin{macrocode}
48 { \color{red} \large}
49 % \end{macrocode}
50 % Am Ende wird alles wieder in den ursprünglichen
51 % Zustand versetzt.
52 % \begin{macrocode}
53 { \normalcolor \normalsize}
54 % \end{macrocode}
55 % \end{environment}
56 % Nun erfolgt noch ein \begin{MeinBeispiel} Beispieltext \end{MeinBeispiel}
57 % wie er durch das Paket \meta{MeinPaket} verändert wird.
58 % \iffalse
59 %</sty>
60 % \fi
61 %\endinput

```

Wenn die Datei kompiliert wird, lädt sie sich selbst zweimal: In Zeile 7 wird aus dem Bereich zwischen `%<*sty>` und `%</sty>` die Datei `MeinPaket.sty` extrahiert. In Zeile 17 wird alles noch einmal eingelesen, was nicht zwischen einem `\iffalse` und einem `\fi` steht (also alles außer Zeile 1–21 und 58–60). Besonderheit bei diesem zweiten Kompilierdurchgang ist, dass der normale Code gewissermaßen verbatim

ausgegeben wird. Alles, was hinter einem % steht, wird als normaler Code ausgeführt. So entsteht die Dokumentations-PDF. Der Bereich um Zeile 17 (genauer: alles, was zwischen %<driver> und %</driver> steht) ist das eigentliche TeX-Dokument, aus dem die PDF erstellt wird.

Gehen wir das Dokument im Einzelnen durch. Wir sehen in Zeile 2 %<batchfile> was mit %</batchfile> in Zeile 9 korrespondiert. Dazwischen steht die Information, welche Code-Blöcke in welche Dateien zu schreiben sind. Es braucht also in unserem Fall keine externe .ins-Datei, da sie in die .dtx-Datei integriert ist.

```

1 %<batchfile>
2 \begingroup % startet eine Gruppe
3 \input docstrip.tex
4 \keepsilent % kein detailliertes Feedback
5 \askforoverwritefalse % überschreibe alte Version
6 \generate{\file{MeinPaket.sty}{\from{\jobname.dtx}{sty}}}
7 \endgroup % beendet die Gruppe
8 %</batchfile>

```

Mit \keepsilent wird unterdrückt, dass Zeile für Zeile ein sehr detailliertes (und nicht wirklich hilfreiches) Feedback in der Kommandozeile ausgegeben wird. \askforoverwritefalse überschreibt vorhandene Versionen einer Datei beim Ausführen der .dtx-Datei. Dieser Befehl sollte gesetzt sein, da man ansonsten bei jedem Kompilieren eine JA/NEIN-Abfrage beantworten muss. Das Makro generate{<Ziel>}{<Quelle>} sorgt für die Verteilung der Code-Blöcke. Fangen wir von hinten an: Alle <sty>-Blöcke (siehe unten) aus der Datei \jobname.dtx werden in die Datei MeinPaket.sty geschrieben.

```

1 %<driver>
2 \documentclass{ltxdoc}
3 \author{Lukas C. Bossert}
4 \title{Mein erstes \LaTeX-Paket}
5 \usepackage{MeinPaket} % lädt das neue Paket
6 \begin{document}
7 \maketitle % erstellt eine Titelseite
8 \DocInput{\jobname.dtx} % fügt sich selbst ein
9 \end{document}
10 %</driver>

```

Der zweite Block (%<driver>...</driver>) beinhaltet alles, was das eigentliche Paket ausmacht. Wir sehen, es ist der Code für ein normales L^AT_EX-Dokument: In Zeile 14 wird das Paket *MeinPaket* geladen, sodass wir es gleich innerhalb des Dokuments nutzen können.

In Zeile 20 beginnt kurz vor dem schließenden `\fi` mit `\<{*sty}` der elementare Code-Block des Pakets. In Zeile 59 wird er wieder geschlossen.

Im zweiten Abschnitt ist auffallend, dass (fast) alle Zeilen auskommentiert sind. Dies ist ein wesentliches Merkmal des Programm-Code-Abschnitts. Text, der als Beschreibung in die Dokumentation kommt, hat ein vorangestelltes `%`. Man kann also die Beschreibung wie gewohnt verfassen, mit `\section{<NAME>}` zur Unterteilung etc., man muss nur auf das `%` am Zeilenanfang achten.

Ein Programm-Code für das Paket wird in die `macrocode`-Umgebung gesetzt. Entscheidend ist die Einrückung von `\begin{macrocode}` um genau vier (4) Leerzeichen nach `%`. Gleiches gilt auch für `\end{macrocode}`. Der Programm-Code selbst wird ohne `%` oder Einrückung geschrieben.

```

1 % \begin{macrocode}
2 \NeedsTeXFormat{LaTeX2e}[2005/12/01]
3 \ProvidesPackage{MeinPaket.sty}
4 % \end{macrocode}

```

Es gibt noch die Umgebung `environment` mit einem notwendigen Argument für den Umgebungsnamen. Der Umgebungsname wird in der Dokumentation in die Marginalspalte gesetzt. Innerhalb der `environment`-Umgebung kann wiederum mit `\begin{macrocode}... \end{macrocode}` gearbeitet werden. Die Beschreibung des Codes ist dabei stets außerhalb der `macrocode`-Umgebung.

```

1 % \end{macrocode}
2 % Es folgt der Code des Pakets. \par
3 % Das Paket "MeinPaket" lädt ein anderes Paket.
4 % \begin{macrocode}

```

So enthält also eine einzige Datei die Anweisungen, wie aus ihr selbst die eigentliche Paket-sty-Datei erstellt werden muss und gleichzeitig die Dokumentations-PDF.

Extraktion von Programm-Code und Dokumentationserstellung

Um nun die Paket-Datei und deren Dokumentation zu erstellen, muss über die Kommandozeile der Befehl

```
lualatex MeinPaket.dtx
```

ausgeführt werden. Das Ergebnis ist die Paket-Datei, wie sie in Listings 3 abgedruckt ist und die dazugehörige Dokumentation (Abb. 1).

Das Beispiel mit dem fiktiven Paket war nur ein kurzer Einblick, was mittels *literate programming* an sich möglich ist. Allen Interessierten sei das Tutorial von Pakin wärmstens empfohlen [8]. Dort gibt es ausführlichere Beschreibungen zu den

einzelnen Befehlen und Hintergrundinformationen, was \TeX -nisch beim *literate programming* passiert.

Fazit

Die drei vorgestellten Möglichkeiten zur Dokumentation und Kommentierung von Code haben Vor- und Nachteile. Es ist nicht nur eine Frage der persönlichen \LaTeX -Kompetenz, eine der drei Varianten zu verwenden, sondern auch eine Frage, des richtigen Einsatzes der Dokumentation- und Beschreibungsart: Anhand von Kommentaren direkt am Code wird dieser für einen selbst und für eine Gruppe mit der man am Code arbeitet auch später noch nachvollziehbar bleiben. Es ist die schnellste und oft auch effizienteste Methode für kurze Code-Sequenzen.

Der Einsatz von Pakten, allen voran `listing`, lohnt sich dann, wenn man das Ergebnis von ausführbarem Code unmittelbar daneben zeigen möchte.

Literate programming sollte vor allem dann zum Einsatz kommen, wenn man viel Code, wie beispielsweise bei einem Paket oder Dokumentenklasse, transparent dokumentieren möchte. Die Effizienz dieser Methode ist beeindruckend, besonders, wenn man dabei noch eine `makefile`-Datei einsetzt, die sich um die Datei-Erstellung und -Verwaltung kümmert.⁷ Bei großen Code-Projekten ist diese literarische Dokumentationsart sicherlich die schönste Form, Code zu lesen.

Listing 3: Programm-Code des fiktiven Pakets `MeinPaket`.

```

1 %%
2 %% This is file `MeinPaket.sty',
3 %% generated with the docstrip utility.
4 %%
5 %% The original source files were:
6 %%
7 %% MeinPaket.dtx (with options: `sty')
8 %%
9 %% IMPORTANT NOTICE:
10 %%
11 %% For the copyright see the source file.
12 %%
13 %% Any modified versions of this file must be renamed
14 %% with new filenames distinct from MeinPaket.sty.
15 %%
16 %% For distribution of the original source see the terms
17 %% for copying and modification in the file MeinPaket.dtx.
```

⁷ Beeindruckend ist dies beim `biblatex`-Stil `oxref` umgesetzt [1]. Zum Einsatz von `makefile`-Dateien siehe Bossert [2].

```
18 %%  
19 %% This generated file may be distributed as long as the  
20 %% original source files, as listed above, are part of the  
21 %% same distribution. (The sources need not necessarily be  
22 %% in the same archive or directory.)  
23 \NeedsTeXFormat{LaTeX2e}[2005/12/01]  
24 \ProvidesPackage{MeinPaket.sty}  
25 \RequirePackage{xcolor}  
26 \newenvironment{MeinBeispiel}  
27 {\color{red}\large}  
28 {\normalcolor\normalsize}  
29 \endinput  
30 %%  
31 %% End of file `MeinPaket.sty'.
```

Dank

Ich danke Florian Claus und Philipp Pilhofer herzlich für Anmerkungen und Korrekturen.

Mein erstes L^AT_EX-Paket

Lukas C. Bossert

March 11, 2020

1 Überblick

Mit dem Paket lässt sich die Schriftfarbe und Schriftgröße innerhalb einer Umgebung ändern.

2 Code-Documentation

Als erstes wird die Grundvoraussetzung für das Paket spezifiziert, anschließend erfolgt die Namensnennung.

```
1 \NeedsTeXFormat{LaTeX2e}[2005/12/01]
2 \ProvidesPackage{MeinPaket.sty}
```

Es folgt der Code des Pakets.

Das Paket "MeinPaket" lädt ein anderes Paket.

```
3 \RequirePackage{xcolor}
```

Es stellt auch eine eigene Umgebung zur Verfügung.

MeinBeispiel Innerhalb der Umgebung (*MeinBeispiel*) wird die Schriftfarbe und -größe geändert.

```
4 \newenvironment{MeinBeispiel}
```

Zu Beginn wird die Schrift rot eingefärbt und vergrößert.

```
5 {\color{red}\large}
```

Am Ende wird alles wieder in den ursprünglichen Zustand versetzt.

```
6 {\normalcolor\normalsize}
```

Nun erfolgt noch ein **Beispieltext** wie er durch das Paket (*MeinPaket*) verändert wird.

Abb. 1: Dokumentation des fiktiven Pakets MeinPaket.

Literatur und Software

- [1] Alex Ball: The Biblalex-oxref package, BibLaTeX styles inspired by the Oxford Guide to Style, Version 2.0.1, 2020, <https://github.com/alex-ball/biblalex-oxref> (besucht am 16. 3. 2020).
- [2] Lukas C. Bossert: »Zur Nutzung von makefile-Dateien«, *Die T_EXnische Komödie*, 31.2 (2019), 64–71.
- [3] Charles Antony Richard Hoare, »Hints on programming language design« in Computer Systems Reliability, (Hrsg.: C. Bunyan), State of the Art Report 20, 1973, S. 193–216, <http://flint.cs.yale.edu/cs428/doc/HintsPL.pdf> (besucht am 6. 9. 2018).
- [4] Donald E. Knuth: »Literate Programming«, *The Computer Journal*, 27.2 (1984), 97–111, DOI 10.1093/comjnl/27.2.97.
- [5] — The T_EXbook, Addison-Wesley, 1984.
- [6] — Literate Programming, CSLI Lecture Notes 27, Cambridge University Press, Cambridge, 1992.
- [7] Brooks Moses, Jobst Hoffmann, Carsten Heinz: The Listings package, Typeset source code listings using LaTeX, Version 1.8c, 2019, <http://www.ctan.org/pkg/listings> (besucht am 16. 3. 2020).
- [8] Scott Pakin: The Dtxut package, Tutorial on writing .dtx and .ins files, Version 2.1, 2015, <http://www.ctan.org/pkg/dtxut> (besucht am 16. 3. 2020).
- [9] Kathrin Passig, Johannes Jander: Weniger schlecht programmieren, O'Reilly Germany, Köln, 2013.
- [10] Geoffrey Poore, Konrad Rudolph: The Minted package, Highlighted source code for LaTeX, Version 2.5, 2017, <http://www.ctan.org/pkg/minted> (besucht am 16. 3. 2020).
- [11] Uwe Post: Besser coden, Rheinwerk Verlag, Bonn, 2018.
- [12] Christine Römer: »Pakete in Deutsch dokumentieren«, *Die T_EXnische Komödie*, 23.2 (2011), 28–36, https://archiv.dante.de/DTK/PDF/komoedie_2011_2.pdf (besucht am 11. 3. 2020).
- [13] Thomas F. Sturm: The Tcolorbox package, Coloured boxes, for LaTeX examples and theorems, etc, Version 4.22, 2019, <http://www.ctan.org/pkg/tcolorbox> (besucht am 16. 3. 2020).